6-5-1984

Dartmouth-Smalltalk: An Exercise in Implementation

Joon Sup Lee
Dartmouth College

DARTMOUTH-SMALLTALK: AN EXERCISE IN IMPLEMENTATION

Joon Sup Lee

Technical Report PCS-TR86-108

DARTMOUTH - SMALLTALK
AN EXERCISE IN IMPLEMENTATION

by
Joon Sup Lee '84

June 5, 1984
Senior Honors Project
Advisor: Mark Sherman
Department: Computer Science

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank professor Mark Sherman and my partner in the project, Phil burrow '86. Phil has committed a large proportion of his time to this project during this term and the progress made in it would not have been possible without his help. I would also like to acknowledge the work he did in preparing the figures in the back of this paper as well as Appendix D.

Professor Sherman has also been an invaluable resource throughout this term. In addition to making his various personal resources, such as literature and terminals, available to us, he has also been always willing to take time out of his busy schedule in order to guide us in the right direction.

INTRODUCTION

Smalltalk-80 is an object-oriented programming language with an interactive environment developed over the last ten years by the Software Concepts Group [SCG] of Xerox's Palo Alto Research Center [PARC].[1] As a language Smalltalk implements many modern features such as dynamic storage allocation and message-passing (call-time binding of procedure names to procedures).

A complete Smalltalk system consists of two major components, the virtual image and the virtual machine. The virtual image is simply the contents of the memory of an operational Smalltalk system, containing all necessary routines and data structures to fulfill the specifications of Smalltalk. The virtual machine is an interpreter program which maintains and manipulates the contents of the virtual image to create a functional interactive system.

In the overall scheme of a Smalltalk system, when a user inputs the source code, it is compiled by the system compiler and the result is a number of so-called Smalltalk objects. All entities, such as procedures, activation records, data structures, etc., which are normally stored in the internal memories of programming systems, are all stored in a uniform fashion in a Smalltalk system and referred to as objects. The compilation process results in two types of new objects being created, the CompiledMethod and Class. A CompiledMethod is

completely analogous to a procedure or a subprogram in ordinary systems and contains Literals, which are simply symbol table information for referencing other objects, and Bytecodes, 8 bit codes that specify the action to be taken by the interpreter. A Class contains the kind of type information usually found only in the symbol table of compilers. However, because of run-time linkage of procedures, Smalltalk must keep up with such information during execution.

After compilation, the CompiledMethods become active through certain commands from the user. When this occurs, the interpreter creates another object called a Context. This object is analogous to a normal activation record, and contains such information as the local variables, instruction pointer, and the runtime stack. The interpreter then uses these three objects to carry out the actions specified by the Bytecodes of the CompiledMethod. It must be noted that the interpreter is not allowed to access the object memory directly. Instead, it must call on the object memory manager to perform any accesses or creation of objects. Analogously, in order to perform any I/O to the user, the interpreter must call on the special I/O primitive routines to do so. Furthermore, it should be apparent that the compiler is merely a CompiledMethod in the object memory. Thus the compilation process itself is carried out through the interpreter.

Figure I contains an overall diagram of these relationships between the various parts of a Smalltalk system. In the diagram,

it can be seen that the term virtual image simply refers to all the objects stored in the object memory, while the virtual machine consists of the interpreter, object memory manager, and the special I/O routines.

In order to encourage the implementation of Smalltalk systems outside of Xerox, the SCG makes a copy of the virtual image available to those wanting to attempt such an implementation. Therefore, a fully operational Smalltalk system may be implemented by creating a virtual machine and loading the virtual image provided by SCG. The formal specifications for such a virtual machine also have been provided by the SCG in the book SMALLTALK-80: THE LANGUAGE AND ITS IMPLEMENTATION, by Goldberg and Robson. Obviously, an alternative to this approach in implementation is to simply start from scratch and build a machine along with the image that will interact with it to behave according to the specifications of a Smalltalk system.

Since its release a few years ago, many groups outside of the SCG have been successful in implementing Smalltalk on various computing machinery, and most, if not all, have taken the first approach and have concentrated on the development of the virtual machine. Although different groups have taken different paths to the creation of a virtual machine, in general, a satisfactory implementation of a Smalltalk virtual machine seems to be a three step process. The first step is a straightforward translation of Goldberg and Robson's formal specifications of the virtual machine into a high-level language. The idea here is to simply

get a system up which meets the requirements for being a Smalltalk system. However, because of the complex nature of Smalltalk, such implementations inevitably prove to be unsatisfactory for application purposes. The problem is that the inefficiencies introduced by a naive translation of the formal specifications into a structured high-level language simply decrease the speed of the system to unusable levels.

Therefore, a second step in the development of an application-oriented Smalltalk system is to optimize the high-level language code. This is done through the usual techniques of open coding, isolation and optimization of special cases, and implemention of more efficient data structures. Depending on the hardware device being used, such improvements may or may not prove to be sufficient to provide a usable Smalltalk system.

In those cases where the hardware does not prove to be powerful enough to adequately support a high-level language version of a Smalltalk system, a second optimization step must be taken. This is usually achieved by programming parts or all of the virtual machine in assembly language or microcode. The state-of-the-art Smalltalk systems currently in existence generally combine assembly language with various other optimization schemes to achieve adequate speed.

The goal of our project was to achieve the first step in this three step process. In other words, we simply wanted to bring up a working Smalltalk system with only minimal

considerations for efficiency, by carrying out a fairly straightforward translation of Goldberg and Robson's formal specifications. For the interpreter and the object memory manager, we chose to use the programming language C on a VAX 11/750 computer with Berkeley 4.2 UNIX operating system. However, for the bitmap display routines, in order to take full advantage of the resources available at Dartmouth, a decision was made to use the newly available MCS68000 based Apple MacIntosh personal computers.

DESIGN DECISIONS AND PROBLEMS ENCOUNTERED

The first major decision in any computing project such as this one is the choice of the implementation language and hardware. Our decision to use C as the language and the VAX 11/750 as the main processor was not based on any performance considerations, but rather was basically forced upon us through availability. The VAX 11/750 was seen as the system least likely to be overcrowded with users. The language C was chosen simply because it was the best-supported language on our UNIX. In contrast, the decision to use the MacIntoshes for bitmap I/O was motivated by two completely differently reasons. In anticipation of the speed problem in Smalltalk systems discussed above, it was hoped that by taking some of the bitmap burden away from the VAX, we would be able to increase performance. In addition, with the anticipation of the influx of a large number of MacIntoshes into

campus, there seems to have been a general eagerness to find as many new applications for them as possible. Although the full impact of this attitude on our decision is difficult to estimate, in retrospect, it does seem to have been a significant influence.

As stated in the introduction, Goldberg and Robson provide the formal specifications that must be met in order to create a Smalltalk virtual machine. These specifications are fairly well detailed, and easy to follow, except for the somewhat frequent errors in printing.[2] In addition, the specifications are in a form of a model implementation written in Smalltalk itself, so in many instances, almost a mechanical translation into C is possible.

At the heart of every Smalltalk virtual machine is the main interpreter. This main interpreter component contains an infinite loop which fetches a Bytecode (the compiled code) and calls a routine to carry out the action corresponding to the value of the Bytecode. These action sequences are described in detail by Goldberg and Robson and does not leave much room for design variations. Therefore, our implementation of the main interpreter closely follows the structure outlined by Goldberg and Robson with only a few minor exceptions.

In the single major area that they leave a design decision up to the implementors, the Method lookup process, Goldberg and Robson state it is possible to lookup every Method being called straight through the message dictionary, but it would be more efficient to utilize an internal table to cache some of the

Methods. In our implementation we have decided to include a Method cache. A Method in Smalltalk is analogous to procedures in most other languages. So a Method lookup is the process by which Smalltalk finds the address of the procedure being called. In normal noninteractive systems, this address determination is done by the compiler, but Smalltalk does not carry it out until the procedure call (message transmission) is actually executed. This is one of the reasons for the existence of Class objects (containing type information) in the runtime environment. The Class objects specify where the message dictionary containing the addresses of all the Methods (procedures) accessible from the current CompiledMethod may be found. By creating a Method cache, we are able to avoid an access to this message dictionary in object memory, with all of its inherent inefficiencies, in many Method lookups. The full effect of this Method cache on efficiency will be discussed later when optimization is examined.

In addition to this design choice, numerous problems were encountered during the implementation of the main interpreter because of the peculiarities of our hardware were inconsistent with Smalltalk specifications. A prime example of this is the representation of floating point numbers. Both Smalltalk and C use 32 bits to represent a floating point number with the first bit being the sign bit and the next 8 bits representing the exponent of 2 and the last 23 bits representing the fractional part of a number between 1 and 2. However, to calculate the exponent in Smalltalk, the 8 exponent bits are taken as an

unsigned number and an offset of -127 is added to it. So the range of exponents is from -127 to +128. C uses the same basic technique but the offset has the value of -128, making the exponent range from -128 to +127. This, in turn, meant that there were numbers which could be represented as a Smalltalk floating point number, but not as a 32 bit C floating point number. This necessitated the use of a 64 bit double-precision floating-point number to do the arithmetic in order to preserve the full range of Smalltalk floating point numbers. Details such as these proved to be very time consuming to code and debug.[3]

Finally, there were many primitives and procedures for which no formal specifications were given by Goldberg and Robson. In these cases, there always seemed to be some guesswork involved in determining what the exact behavior of the routine should be.[4] But overall, as was mentioned earlier, Goldberg and Robson's specifications of main interpreter are complete and well thought out and provide an invaluable tool for anyone implementing the system.

In contrast to the specific detailed description of the main interpreter, Goldberg and Robson's specifications for the object memory are much looser, requiring more design initiative of the implementor. All that is required of the object memory is that its interface with the main interpreter behave as specified. The object memory is simply the space where all objects of a Smalltalk system are stored, and is organized as a heap. The objects in the heap are maintained by the object memory manager.

Furthermore, the organization of the object memory is such that only the object memory manager itself may access the heap directly. All other components of the system, mainly the interpreter and the I/O routines, can only access the heap indirectly through the object table. The object table is a 64k word (16 bits) segment of contiguous memory.[5]

An object pointer is a 16 bit word, which serves as an indirect reference to an object in the heap or as a representation of a small integer.[6] When the least significant bit of the object pointer is 0, the unsigned value of the pointer serves as an offset into the object table to specify a 2 word segment block. The 32 bits of this block specify various information about an object including a 20 bit address of its position in the heap. When the least significant bit of an object pointer is 1, it is taken as a representation of a small integer, whose value is the 2's complement value of the first 15 bits (See Figure II). Thus small integers in the range of -2^14 to +2^14 - 1 are given special treatment here.

This scheme of indirect addressing using a 16 bit object pointer has several interesting properties associated with it. First of all, it has the advantage of significantly increasing the number of objects that may be referenced at any given time. Since Goldberg and Robson claim the average size of an object in a functioning Smalltalk system is 10, the number of referenceable objects on the average using a direct address scheme with a 16 bit pointer is (2^16)/10. However, using our indirect scheme, we

are able to reference 2^15 objects no matter what size they are and this can be easily increased to 2^16 by not representing small integers as a special case of the object pointer. Note that if this modification were made, the size of the object table would have to double and the value of the object pointer times two would serve as the offset into the table. These modifications point out the advantages of the current scheme of using the least significant bit as a flag for small integers. If the object pointer is taken to be a 16 bit unsigned integer, then small integer representations are all odd numbers and the object pointers with entries in the object table are all even numbers. Since each entry in the table is two words long, we are able to reference every block by using the integer value of the pointer as an offset into the table.

Another important ramification of the object pointer referencing scheme is that there are now two ways to run out of memory. Not only can we run out of heap space by having it full of objects, we can also run out of object pointers if we have a lot of small objects. In the latter case, there may still be a lot of free heap space but there would be no way to reference it. In our implementation, the maximum object memory space is 2^20 words, 64k of which will be used for the object table. With an average object size of 10, this means there can be on the average (2^20 - 2^16)/10 objects in the heap. Since we only have 2^15 object pointers and (2^15) < (2^20 - 2^16)/10, we will most

likely run out of object pointers long before we run out of heap space.

Because objects are constantly being created and destroyed in a Smalltalk system, the use of a heap and object table requires the maintenance of free blocks in the object table and free chunks of memory in the heap. Both of these free memory collections are maintained as linked lists. All free pointer blocks are contained in a single list while free chunks of memory are kept in several lists according to the size of the free chunk.

The free pointer list uses one of the words in each free pointer block to store the object pointer for the next free pointer. In the free chunk lists, each chunk still has an object pointer associated with it. Thus this list is organized so that the second word of the object in the heap contains the object pointer of the next chunk in the list. The first word of the object contains the size of the chunks in number of words (See Figure III). The end of both of these lists are marked by a unique small integer representation of an object pointer (65535 = 2^16 - 1 which has all bits set) in the link field.

Although the maximum size of the heap space is limited to 2^20 words by the organization of the object table, there are still many different ways to organize this space. One possibility, suggested by Goldberg and Robson, is to segment the heap space into 16 segments of 64k words each. In this case, the first 4 bits of the 20 bit address would specify the segment and

the remaining 16 bits would specify the address within the segment. However, in our implementation, we decided to forgo the segmentation scheme and allocated the heap space as one contiguous chunk of memory containing 2^20 words. The last 64k words in this space is used as the object table.

Originally, we had planned to follow the segmented scheme in our implementation. However, once the copy of the virtual image to be loaded onto the system was received from Xerox, it was found to be organized as a single contiguous block. Since a segmented scheme would have involved a rather complex relocation of the objects to insure no objects crossed segment boundaries, or a special referencing scheme to allow correct handling of objects crossing segment boundaries, we decided to adopt the unsegmented scheme in order to facilitate the loading process. We also realized that this decision, although not based on performance considerations, would have a significant impact on the behavior of our Smalltalk system. Therefore, it was decided that the supporting structures for the segmented implementation should be left intact in the object memory routines. The result is that only minor modifications will be necessary to change to a segmented heap should such a system be found more desirable than the current scheme.

As was mentioned briefly above, Smalltalk objects are not permanent entities. Therefore, an integral part of memory management is the allocation and deallocation of objects in order that memory space may be reused after an object is destroyed.

Given a heap organization of memory, allocation can be handled simply by keeping a linked list of free chunks of memory. We may then satisfy allocation requests by returning a pointer to the first chunk found in a linear search to be large enough to meet the request (i.e. first-fit). However, if the free chunk is larger than the size actually requested, it would be a waste of space to allocate the whole chunk. In such cases, the free chunk is split into two parts. The first part, which is exactly the size of the request, is allocated to be used, while the remaining space is returned to the free list. Since both the linear search of the free list and the splitting of large chunks takes precious processor time, and given that most objects in a Smalltalk system are rather small (average size = 10 words), our implementation actually uses 20 linked lists to keep up with the free chunks. The free chunks are separated into the individual lists according to their size, with all chunks larger than 20 words going into the last list. The result is increased speed in allocation because we have eliminated the need for a search in allocation of chunks less than 20 words in size. Moreover, this efficiency gain is achieved at the minor cost of storing 19 extra pointers to the heads of lists.

During normal operations, the allocation and deallocation of objects occur randomly in a Smalltalk system. As a result, fragmentation of the heap space may occur. Fragmentation is a phenomenon where the free spaces in a heap do not exist as one contiguous block, but rather as many smaller blocks separated by

allocated chunks of memory. The result is that an allocation request may not be met even though the total amount of free space exceeds the size of the allocation request because there is no single chunk large enough to satisfy the request. The solution here is a process known as compaction, where all the allocated chunks of memory are physically moved to one end of the heap space to create one large free chunk of memory. Note that, since we have no explicit coalescing mechanism, compaction also serves to coalesce any neighboring free chunks into a single, larger free chunk. However, since the objects in memory are physically moved, compaction requires that all pointers to these objects be updated. This can often be an overwhelming problem in systems where direct reference to memory is allowed, but since we only allow indirect references to the heap space through the object table, we are required to update only one pointer (20 bit address in the object table) for each object.

Unfortunately, our memory manager has no explicit detection mechanism for fragmentation and simply carries out compaction whenever an allocation request cannot be met by simple search through the free chunks list. The behavior of such a compaction routine will depend heavily on the organization of the object memory. If the heap space is segmented, each individual segment can be compacted independently. As a result, each individual compaction will take a shorter amount of time than in a unsegmented implementation with same object memory size. However, it will also be true that compaction will occur more

frequently since a 64k heap will suffer fragmentation quicker than a 2^20 word heap. This presents the designer of the system with a rather simple tradeoff between short, frequent interruptions and long, periodic ones. Ultimately, the desirability of either of these choices will depend on the hardware and the size of object memory. For our implementation, because of the large size of the object memory, it seems reasonable to assume that a full compaction of the whole heap will take a noticeably long time and thus the segmented heap scheme will probably provide a more desirable behavior.

It is also possible to avoid the long pause of a full compaction while keeping the unsegmented organization of the object memory by employing a more sophisticated compaction algorithm. Instead of carrying out a full compaction every time, the compaction routines could be modified to compact only within a certain subrange of the object memory. However, this scheme also requires a more sophisticated detection mechanism for fragmentation, since it must be able to detect when a certain subrange has become fragmented. Although many different criteria could be used for this detection, most simple ones will probably involve counting the number of very small free space chunks within the subrange and carrying out a compaction whenever this count is above a certain value or simply keeping track of the time elapsed since the last compaction and carrying out a compaction after a specified period. By employing such a compaction algorithm, we can duplicate the more frequent, shorter

interruption behavior exhibited by the compaction of a segmented heap.

We could improve the behavior of compaction even further by employing a super-sophisticated compaction technique such as parallel compaction, where an independent compaction process carries out a compaction asynchrously while the main interpreter is still running. Such a technique involves many complicated safeguards to insure against inconsistencies created by simultaneous access to a single object by both the main interpreter and the compactor. However, given our knowledge that most objects in Smalltalk are very small, and our relatively large object memory, we do not anticipate fragmentation to be a large problem and either segmentation or subrange compaction will probably prove to be sufficient to give acceptable performance. Therefore, the implementation of such sophisticated techniques as parallel compaction would not justify the extra complications and the programming effort required.

As a final note on this subject, it should be realized that in addition to creating a single large chunk of free memory, compaction also serves the important function of freeing up object pointers in our system. Recall that our organization of the free chunks list requires an object pointer for every free chunk in the list (Figure III). Therefore, by decreasing the number of free chunks in the system, compaction frees many object pointers for use in allocation of new objects. Since our calculations above showed we were more likely to run out of

object pointers than object memory space, this recovery of object pointers is an important side effect of compaction.

In an interactive system such as Smalltalk, the problem of deallocation proves to be considerably more complex than allocation. In general, there are two basic deallocation schemes in a computing environment, explicit and automatic. In explicit deallocation, the programmer must specify when storage is to be deallocated. In such a system, when an object is prematurely deallocated, pointer use will cause a system error, which requires the restart of the program. Although this inconvenience may be acceptable in a noninteractive environment, where the programmer can fix the bug and reload the program, it is clearly not viable in an interactive system since a system error could mean the loss of a large amount of work.

As a result, interactive systems must provide some type of automatic deallocation. In such systems, objects which are no longer accessible are identified automatically and deallocated. An inaccessible object is an object in memory which has no pointers referring to it. Although there is a wide variety of automatic deallocation systems in existence, most can be thought of as following one of the two basic Methods, reference counting and marking garbage collection (Also called scan-and-marking and mark/sweep garbage collection).[7] In a reference counting scheme, for each object in memory, an explicit count of the number of pointers currently pointing to it is maintained (In an 8 bit field in the object table block in our implementation. See

Figure II). If this count ever reaches zero for an object there are no more pointers to the object and it is thus inaccessible to the system and can be deallocated.

This approach has been shown to have two basic weaknesses. The first is it involves a tremendous overhead in terms of processor time. Since a reference count is kept for every object, it must be updated every time a pointer is stored. In currently operating Smalltalk implementations, this overhead has been shown to have a significant impact on performance because pointer storage is such a high frequency operation [Wirfs-Brock, 1982][Deutsch, 1983].

An additional drawback of the reference counting approach is it may not be able to identify all inaccessible objects if cyclic structures appear. A cyclic structure occurs when 2 or more objects point to each other, thus making their reference counts greater than zero. However, if there exists no other pointers to these objects outside the cycle, then they are inaccessible to the system, but they will not be deallocated through a reference counting technique since their reference counts will still be greater than zero.

In order to detect and deallocate such structures, a second Method, called marking garbage collection, must be used. The idea here is to start at some root pointers, usually the pointers contained in the registers of the main interpreter and mark all objects accessible from it. Note that this is a recursive traversal technique whereby all objects directly referenced by

the root pointers are marked and then all those referenced within the newly marked objects are marked and so on until all accessible objects have been marked. After this marking process, the object table is searched and all unmarked objects are deallocated. The major disadvantage of this approach, like the reference counting technique, is the processor overhead involved. However, the form of this overhead differs significantly between the two systems. As explained above, the overhead of a reference counting system is incurred incrementally, in the form of a couple of instructions for each storage operation. In contrast, marking garbage collection is performed periodically, and the overhead occurs all at once. Thus a user of a reference counting system will notice a general slowness in the system while a user on a marking garbage collection system will note that his system will have to completely suspend operations periodically in order to perform deallocation.

In our implementation, we have decided to utilize both Methods of deallocation. Reference counting is used for normal incremental deallocation until there is not enough space due to cyclic structures. At this time a marking garbage collection is performed to recover inaccessible cyclic structures. The garbage collection actually is initiated by the allocation routine, which has the following structure: When an allocation request is made, the free chunk lists are searched in an attempt to meet the request. If this is not possible, then a compaction is performed. If this still does not produce enough free space, a

marking garbage collection is performed and the memory is recompacted if necessary. If this still does not produce enough space, the system has simply run out of memory and must halt.

In addition to this combination approach, our implementation uses an efficient traversal routine to save space and time. In both of the deallocation schemes, pointers must be traversed at some time. In a reference counting scheme, when an object's count reaches zero and it is being deallocated, the reference counts of all other objects accessible from this object must be decreased and if any one of these reference counts reach zero, this process must be repeated. In a marking routine, pointers must be traversed in order to mark all objects indirectly accessible from the root object pointer. Although both of these traversals are logically recursive procedures, in order to avoid the inefficiencies in the implementation of recursion, we use a nonrecursive technique.

We achieve this by reversing the pointers as we traverse downward toward the original object. For example, assume that the i-th word of object A points to object B whose j-th word points to object C whose k-th word points elsewhere, so that we have $A_i \rightarrow B_j \rightarrow C_k \rightarrow ?$. When we have traversed downward past object C, our pointers will be such that the k-th word of C points to B whose j-th word points to A whose i-th word will point to its ancestor or will be a Non Pointer (value 65535) if A is the root of this traversal. So we have $\rightarrow C_k \rightarrow B_j \rightarrow A_i \rightarrow$Non Pointer. The only complication is that we must not only remember which object

we came from, but exactly which word. Therefore, we must store the indicies i,j,and k somewhere. But this is resolved because these traversals only occur during deallocation and marking, when the reference counts are about to be reset, so we simply store the index in the reference count field of the object table block. The two possible situations during a traversal described here is shown in Figure IV. This iterative procedure not only saves the processor time which would have been required for process switching in recursion, but also avoids the stack space needed for recursion by using reversed pointers to record the traversal path.

The decision to combine both deallocation Methods in our implementation was motivated by a desire to make full use of the available object memory space. In general, reference counting is the preferred approach where a large memory space is available. This is because a marking collection on such a memory would take a prohibitively long time and the space wasted by cyclic structures would not be significant relative to the size of the memory. In contrast, marking garbage collection is seen as more suitable for smaller memory spaces in which the collection can be performed in an acceptably short period of time and the space saved by deallocating cyclic structures will be significant.

Given the relatively large size of the memory space (2^20 words), a reference counting deallocation scheme alone probably would have been adequate in our implementation. In fact previous implementors have found that cyclic structures do not pose a

problem at all [Ingalls, 1978]. However, our system gains the added safety of a marked collection system without incurring any significant penalties. In fact, the only penalty is the additional memory space of our processor taken by the code of the garbage collector because such a collection is only performed when all other Methods for allocating enough memory have failed.

FUTURE IMPROVEMENTS AND PERFORMANCE OPTIMIZATIONS

It was emphasized above that straightforward, high-level language implementations of Smalltalk are invariably too slow to be useful. Thus in order to develop an application-oriented system, extensive optimization must be carried out to increase the performance of the system. Although our implementation was carried out with only a minimal consideration for performance, a couple of optimizing features nevertheless have been incorporated into our system.

The most significant optimizing feature is the Method cache for the Method lookup process described above. This easy-to-implement feature, while saving only a couple of memory accesses for each message lookup, proves to be significant because of the frequency with which a Smalltalk system performs the Method lookup. A group at UC Berkeley carried out performance comparisons between two implementations of Smalltalk, one with the Method cache and the other without it [Ungar, 1982]. They found that on the average, a Method lookup without a Method

cache took 10 times as long as a lookup with the cache. More importantly, they claim that the performance of the system as a whole is 40 percent slower without the cache. Since these tests were carried out on a Smalltalk system written in C running on a VAX 11/780, we can reasonably assume that the optimizing effects of the cache on our system will be of a similar magnitude.

The second optimization technique used in our implementation is the nonrecursive traversal of the objects during garbage collection. As explained above, this saves the overhead involved in recursive routines. Although we cannot expect this optimization to have the impact of the Method cache, it should still provide a noticeable improvement in performance.

These two improvements alone make impressive gains in the attempt to bring the speed of a Smalltalk system up to a usable level, but clearly, there is yet much room for further optimization. Although it was not part of our original goal to create a fully optimized Smalltalk system, it is nevertheless important to examine the optimizations that are possible. Since we have no first hand experience on the subject, we will carry out this examination by studying the techniques other Smalltalk implementors have utilized to increase the performance of their systems.

It has probably come to the attention of the reader that almost all of the design decisions discussed so far, especially those concerning performance of the system, have been in the object memory manager component of the virtual machine. This is

also the area of Smalltalk implementation which has been targeted most often by other implementors for optimization. This is a logical choice when one considers the fact that some implementations of Smalltalk have been shown to spend as much as 70 percent of the processor's time executing memory management routines [Wirfs-Brock, 1982]. Furthermore, most of this time is spent simply maintaining the reference counts for automatic deallocation. For example, Suzuki claims 80 percent of the execution time for pushes and pops onto a stack is spent on reference counting [Suzuki, 1983]. In light of this dominance of processor time by reference counting routines, numerous techniques have been suggested for speeding up the reference counting routines and/or avoiding reference counts altogether whenever possible.

In the straightforward implementation of a Smalltalk virtual machine, an update on reference counts occurs every time a pointer is stored. Since this storage process could be overwriting some other object pointer, the reference count of the object corresponding to the old value in this location is decremented. By storing an object pointer, we are also creating a new reference to an object in memory and therefore, the reference count of this object must be incremented. In addition, because our implementation uses only an 8 bit field in the object table block to store the reference count, an overflow may occur. In order to handle such cases, reference counting routines check to see if the value of a reference count is greater than or equal

to 128, and if it is, the reference count is neither decremented nor incremented. Thus once a reference count reaches 128, it remains there until a marking garbage collection occurs.

The most effective techniques for optimizing reference counting currently implemented identify special situations in which the reference count need not be updated during a pointer storage and avoid reference counting in such situations. Although different implementors have identified different sets of these special situations, the following three examples should give a general view of this optimization technique. The first example, a method called "deferred reference counting", is based on the observation that many object references from a stack are transient in nature [Wirfs-Brock, 1982]. For example, when assigning an object pointer to a variable, the pointer is first pushed onto the stack, increasing the reference count by one, and then later popped from the stack, decrementing the reference count, before it is finally stored into the variable, again increasing the reference count by one. So the net effect on the reference count of the object is to increase it by one but three updates were performed to achieve this. Since this push, pop, and store process is the basic action sequence of the "store instance variable" Bytecodes, by performing no reference counting on the objects on the stack during the execution of these Bytecodes, we are able to decrease the reference counting overhead.

A similar situation arises when a Method returns the top of the stack. In this situation, the object pointer on the top of the currently active Context's (the callee Context) stack is pushed onto the stack of the caller's Context, increasing the reference count by one. But when the return is finished and the callee Context is being freed, the reference count of the returned object pointer will be decremented. Therefore, the net effect of this return process is that the reference count stays the same. We can thus avoid two reference count updates by explicitly destroying the object pointer from the callee Context's stack by storing a nonreference-counted object such as Nil (A special object pointer signifying no object) over it immediately after pushing the return value onto the caller's stack [Ungar, 1982].

As the above example shows, certain special objects, especially Nil, are nonreference-counted. In reality this means that their reference count is set to 128 to indicate overflow and thus an attempt to update their reference count is still made when they are involved in a store pointer operation. Therefore, a call to a reference counting routine can be avoided when the system can guarantee that the object pointer being written over in a pointer storage operation is a nonreference-counted object. Analogously, when the system can guarantee the pointer being stored is a nonreference-counted object, reference count incrementation can be avoided [Ungar, 1982]. The most common application of these guarantees occurs in the allocation and

initial assignment of new objects. The specifications of the object memory manager require the fields of a newly allocated object to be initialized with the object Nil. Therefore, reference count incrementation can be avoided during the initialization of an object about to be allocated and reference count decrementation can be avoided when storing pointers into a newly allocated object.

Obviously, these three cases do not exhaust the situations where reference counting may be avoided. However, as the situations become more specific, the performance impact of avoiding reference counting in such situations becomes less significant. Eventually, a point will be reached where the performance rewards do not justify the extra programming effort required to implement the optimization. The exact number of special situations that may be optimized before this point of negative return is reached will depend on the specific details of the individual implementation projects. As a final note on reference counting avoidance techniques, it is encouraging to see various groups that have implemented these optimizations report being able to avoid between 70 to 85 percent of the reference counting required for a straightforward implementation [Suzuki, 1983] [Ungar, 1982].

Obviously, no matter how good these avoidance techniques are, some reference counting always will be necessary in a Smalltalk system. Therefore, many implementors have found it worthwhile to employ techniques to speed up the actual updating

process. For example, a UC Berkeley group changes the format of the object table in order to eliminate overflow checking. As explained above, an overflow check must be made before every update of a reference count. However, by increasing the size of the reference counter from 8 to 32 bits this group eliminates the possibility of an overflow [Ungar, 1982]. Note that this gain in speed is achieved at the cost of an increased size for the object table and therefore, this technique is not really practical for systems where memory space is at a premium.

Another aspect of a Smalltalk object memory manager which takes up a lot of processor time is the indirect addressing scheme using the object pointers. Since the interpreter is allowed to manipulate only the object pointer and not the actual address of an object in the heap, every access into an object involves a level of indirection. This problem is exacerbated by the fact that the 20 bit heap address stored in an object table block is really an offset from the beginning of the heap and not an actual physical address. Therefore, in reality, the machine must first calculate a physical address from this offset, thereby adding another level of indirection, before an actual access to the heap can be made. In order to remove this indirection, many implementors have modified the object table so that the address field of each object table block holds the actual physical address of the object in the heap [Wirfs-Brock, 1982]. Unfortunately, like the expansion of the reference count field, this optimization usually involves increasing the size of the

object table block and thus is not suitable for small memory machines.

The final aspect of the object memory manager usually targeted for optimization is the allocation of Contexts. In theory, all objects in Smalltalk are to be treated identically by the object memory manager, whether they are CompiledMethods (procedures), Contexts (activation records), or data structure representations. However, many researchers have found that significant performance gains may be made by treating certain type of objects in a special way. For example, studies have shown that 80 percent of all objects allocated in a Smalltalk system are Contexts [Suzuki, 1983]. Moreover, of the two possible sizes for Contexts, 18 and 28 words, the smaller size Contexts are allocated much more frequently. Therefore, by writing specially optimized routines to handle the allocation of these Contexts, a significant improvement in performance can be achieved. These schemes employ such techniques as keeping a special free chunk list for Contexts, dedicating a segment in heap just for the storage of Contexts, or even allocating the Contexts in a stacklike manner, similar to the way activation records are allocated in a normal programming environment [Wirfs-Brock, 1982] [Ungar, 1982] [Deutsch, 1983].

Although, as noted before, the object memory manager tends to dominate the processor time in a Smalltalk system, it is by no means the only area where optimization may be achieved. many improvements to the interpreter component have been shown to have

significant impact on the performance of the system as a whole. In addition, most of these improvements employ such conceptually simple and common techniques as open coding. For example, our implementation handles the dispatching of Bytecodes by successively going through several branches, each of which divides the 256 Bytecodes into smaller groups. Thus to fully dispatch a Bytecode, several procedure calls must be made. These procedure calls are unnecessary and can be eliminated by employing a single 256-way branch to decode a bytecode completely. In addition, we can further open code the action sequence of each Bytecode so that most Bytecodes can be handled with only a single subroutine call. Improvements similar to these were implemented by the UC Berkeley group and it was found that the unoptimized version was 2.5 times slower overall than the optimized version [Ungar, 1982].

CURRENT STATUS OF THE PROJECT

In the introduction to this paper, we stated there were three major components to our Smalltalk virtual machine, the interpreter, object memory manager, and the I/O routines. However, up to this point, only the interpreter and the memory manager have been discussed while the I/O routines have been completely ignored. The reason for this omission is that these routines have not yet been implemented. The status of the project at the writing of this paper is that we have a fully

operational interpreter and an object memory manager, but there is no I/O.

Since there is no real output from our program yet, the interpreter and the memory manager were debugged by comparing the traces produced by our program to the sample traces received from Xerox. These traces follow the details of memory accesses, Bytecode sequences, and procedure calls and returns for the first 1900 Bytecodes carried out when the virtual image is loaded. Although the fact that we can interpret the first 1900 Bytecodes correctly is no proof of complete correctness, we can reasonably assume that the basic design of our implementation is sound. Therefore, we have relatively complete implementations of two of the three components of a Smalltalk virtual machine.

The original plans call for the use of the Apple MacIntosh personal computers to carry out the bitmap display necessary for I/O in Smalltalk. Our failure to implement this portion of the project can be attributed to a combination of several factors. The first and the most important barrier was simply the lack of time. This project was taken on as a part-time endeavor by two students (i.e. one student was also taking classes and the other was involved in other time consuming activities), with the help of a faculty advisor. It has taken 12 weeks and between 250 and 300 hours of each of the two student's time to reach this stage of the project, which consists of approximately 6000 line of code in C. Unfortunately, some of this time, approximately the first two weeks, was spent on learning C and the UNIX since neither

student had been exposed to them before the project. This shortage of time was further exacerbated by the late arrival of the MacIntosh equipment and software. The complete development software for the MacIntosh was not made available to us until about four weeks from the deadline for the project.

Another significant barrier to our completion of the I/O routines was the lack of correct documentation for the MacIntosh. The documentation for the MacIntosh has been found to be so sparse and/or contradictory that such simple operations as opening the serial I/O ports have proven almost impossible. Apparently, this is a universal problem and not unique to Dartmouth as a search through the news network has shown numerous messages complaining about the inaccuracies and inadequacies of MacIntosh documentation.

FUTURE PLANS

Obviously, next step in this project is to implement the I/O routines to create a completely operational Smalltalk system. Although it is impossible to state exactly how much effort this will embody, taking the difficulties explained above into consideration and in light of the fact that the I/O routines, consisting of about a dozen routines, is the smallest of the three major components of the virtual machine, we would guess about two more weeks of full time work on the project should prove to be sufficient.

Even after the completion of the I/O routines, the sluggishness of our system will probably necessitate the implementation of at least some optimizations discussed in the previous sections. Although the decision of which optimizations to implement will depend on a variety of factors, using the ease of implementation and the significance of the impact on performance as the main criteria, we recommend that they be implemented in the following order until an acceptable performance level is achieved:

1) Implement segmentation of the heap space.
2) Utilize open-coding to speed up all facets of the virtual machine.
3) Modify the organization of the object table to be able to eliminate overflow checking of reference counts and to be able to store physical address directly (Both the address and the reference count fields will have to be expanded to 32 bits each).
4) Implement as many reference counting avoidance techniques as feasible.
5) Specialize Context allocation routines.

For those interested in implementing these future plans, the section, GETTING AROUND OUR SMALLTALK has been included to ease the understanding of currently existing code.

PERSONAL REFLECTIONS ON THE PROJECT

In looking back on this project experience, I am filled with mixed emotions. On one hand, there is a sense of disappointment and frustration at not having completed the project. I remorse the fact that I did not have enough foresight to start this project sooner to allow myself more time to finish it. However, on the other hand, I also cannot help but feel that it has been a worthwhile learning experience.

The project itself provides almost a perfect ending to an undergraduate computer science education. Although it does not really involve any original research, the project combines many major concepts studied in various computer classes to create a unified system. The gamut of subjects covered by this project ranges from such operating system topics as heap storage management and automatic deallocation of objects to the high-level language topic of run-time versus compile-time binding of procedure addresses. In a sense, it is nice simply to see how these various seemingly unrelated topics studied in different classes interact in a real-world system such as Smalltalk. Therefore, taken as a learning experience, I must conclude that it was a somewhat disappointing but definitely worthwhile experience.

GETTING AROUND OUR SMALLTALK

The best way to approach this program is to read Goldberg and Robson's book SMALLTALK-80: THE LANGUAGE AND ITS IMPLEMENTATION. Special attention should be paid to the section on implementation and the rest of the book may be skimmed. Our program follows the structure of Goldberg and Robson's model implementation closely for the most part and any differences between the two may be found in Appendix C. Appendix A and Appendix B should also prove helpful in understanding some of the details of our implementation. Appendix D lists all the procedure and global names used by the book and our implementation and should prove to be an invaluable tool for those trying to compare the two sources.

The debugger found in the comptest routine of our implementation allows the examination of the object space and various runtime registers. The following are the options allowed:

1: PEEK - Allows viewing of object memory. Input the segment number (really the value of the first 4 bits of the 20 bit heap address since our implementation is unsegmented), address within the segment (the last 16 bits), and the number of words you want to look at. You will stay in the PEEK mode until you input 0 as the number of

words to be read.

2: STACKVALUE - Prints out the content of the stack of the currently active Context. Must input the offset from the top of the stack for the position to be read.

3: TRACE - Resets the trace value. A value above 4 suppresses all tracing. 1 will reproduce the tracing in tape file 4, 2 will reproduce tape file 5, and 3 will reproduce tape file 6. Our tracing is slightly more crude than the sample from Xerox and will not print out the string names of messages and objects. Instead, we simply print out the value of the object pointers.

4: POKE - Allows storage of any value into any space in the object memory. First 2 inputs specify the address the same way as in PEEK. The third input is the unsigned integer to be stored, and the last input is the continue option. You will remain in the POKE mode until a 0 is entered for the continue option.

5: DIAGNOSE - prints out the current values of various run-time registers.

6: EXIT - Exit debugger and resume execution.

7: JUMP - Jumps the number of instructions specified by the input.

The source code is organized in the manner analogous to Appendix D with a file corresponding to each of the chapters in the implementation section of Goldberg and Robson's book.

FOOTNOTES

[1]The number appearing after the name Smalltalk specifies what year that particular version was released. For example, Smalltalk-72 was the first version released in 1972 and Smalltalk-80 is the current version released in 1980. All references made in this paper are to Smalltalk-80 and thus the number will be omitted.

[2]Refer to Appendix A for a list of misprints identified in the specification of Smalltalk-80.

[3]Refer to Appendix B for a list of implementation details which proved to be troublesome in our project.

[4]Refer to Appendix A for ambiguities found in the specification of Smalltalk-80.

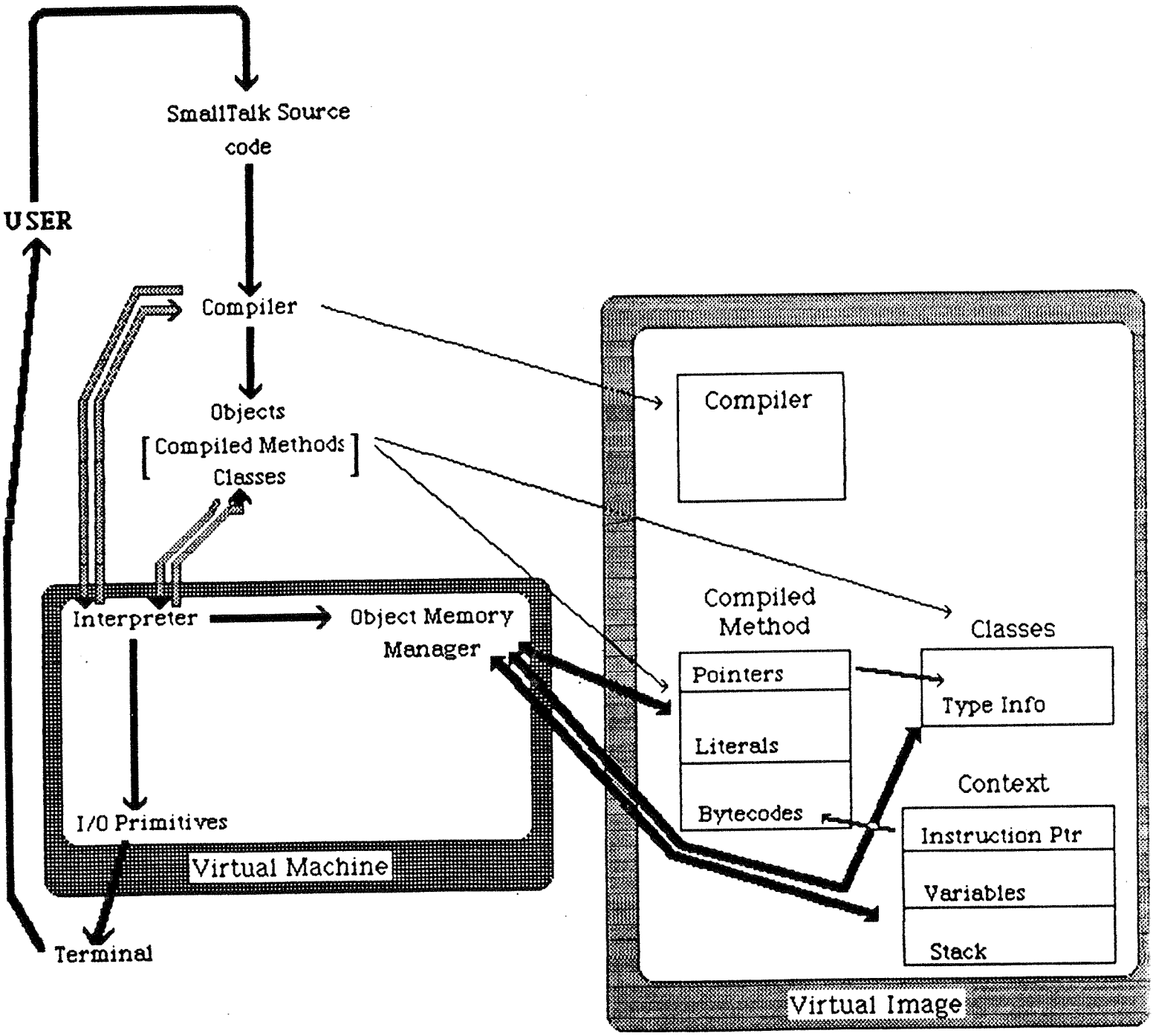
[5]All references to a WORD in this paper are talking about a 16 bit word. Consequently, a byte is 8 bits.

[6]In many Smalltalk literature the object pointer is referred to as an Oop. We will continue to refer to it as the object pointer in this paper.

[7]We choose to use the term marking garbage collection to refer to this technique throughout the paper because it is the term used by Goldberg and Robson.

Figure 1 :

Interface within a SmallTalk System

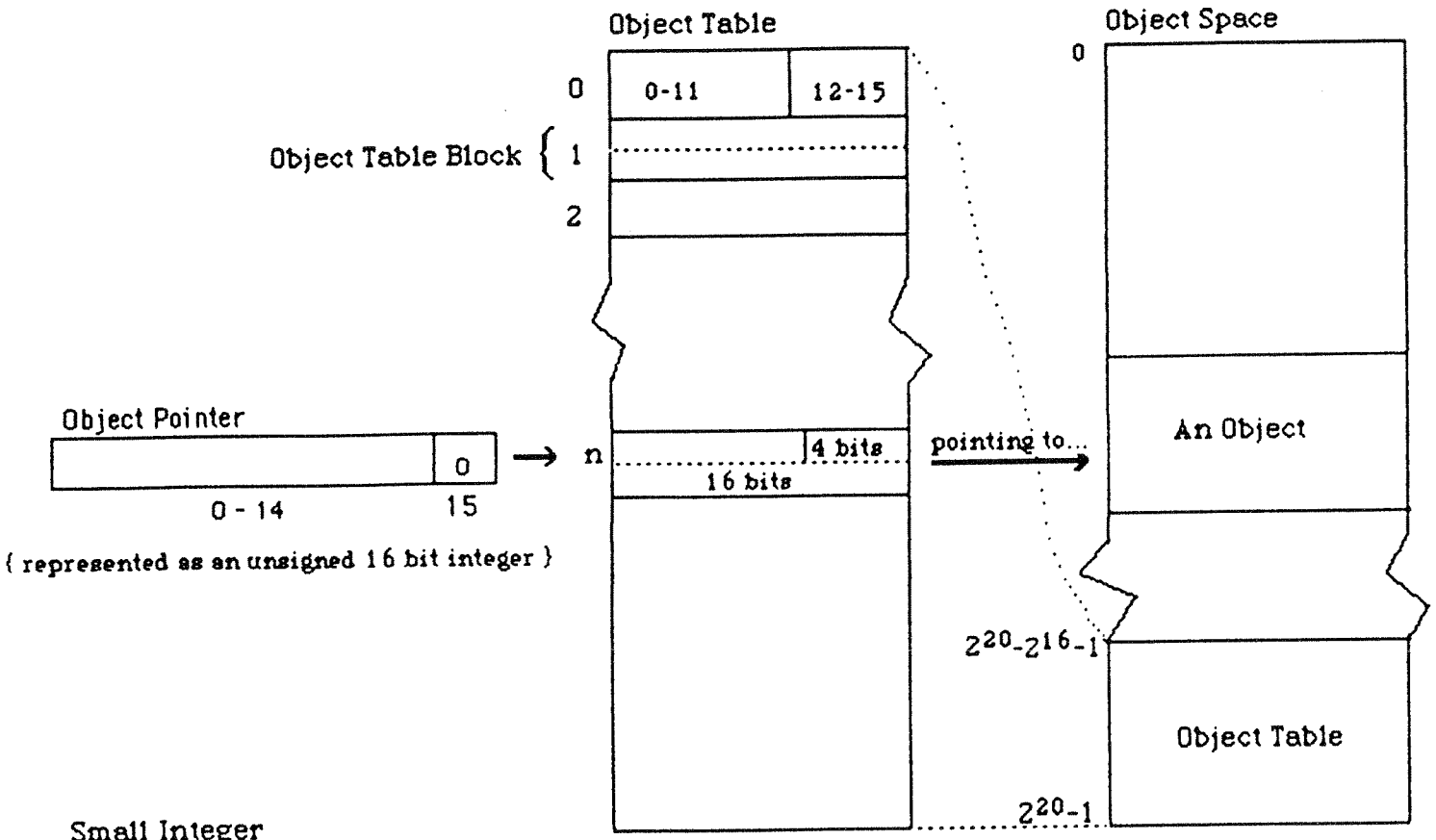


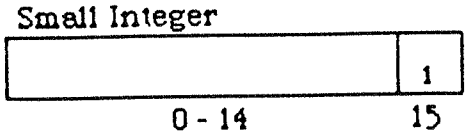
Key:

———	: show that these elements actually exist in the object memory
———	: actual pointers within the object memory
≈≈≈≈≈	: these objects interact with the interpreter to carry out actions
▬▬▬	: flow of control

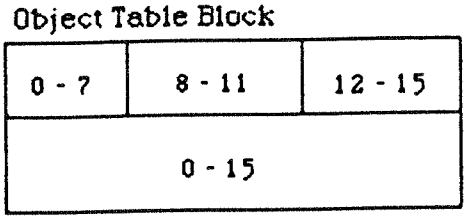
Figure 2

Indirection through Object Table and Small Integer Representation





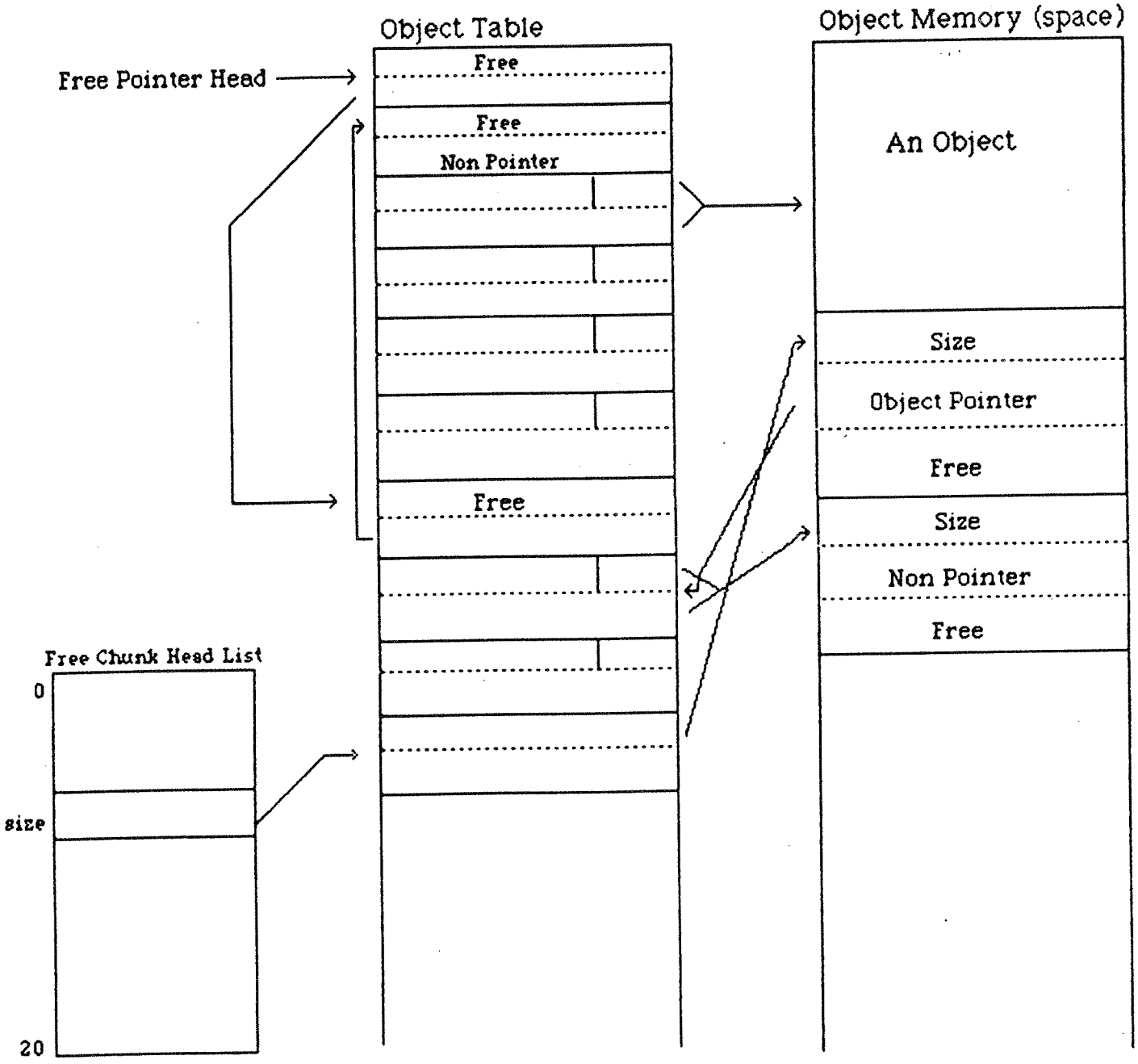
{ taken as two's compliment
to get integer value }



0 - 7: The reference count.
12 - 15
0 - 15 : 20 bits form address of the object in the heap

Figure 3

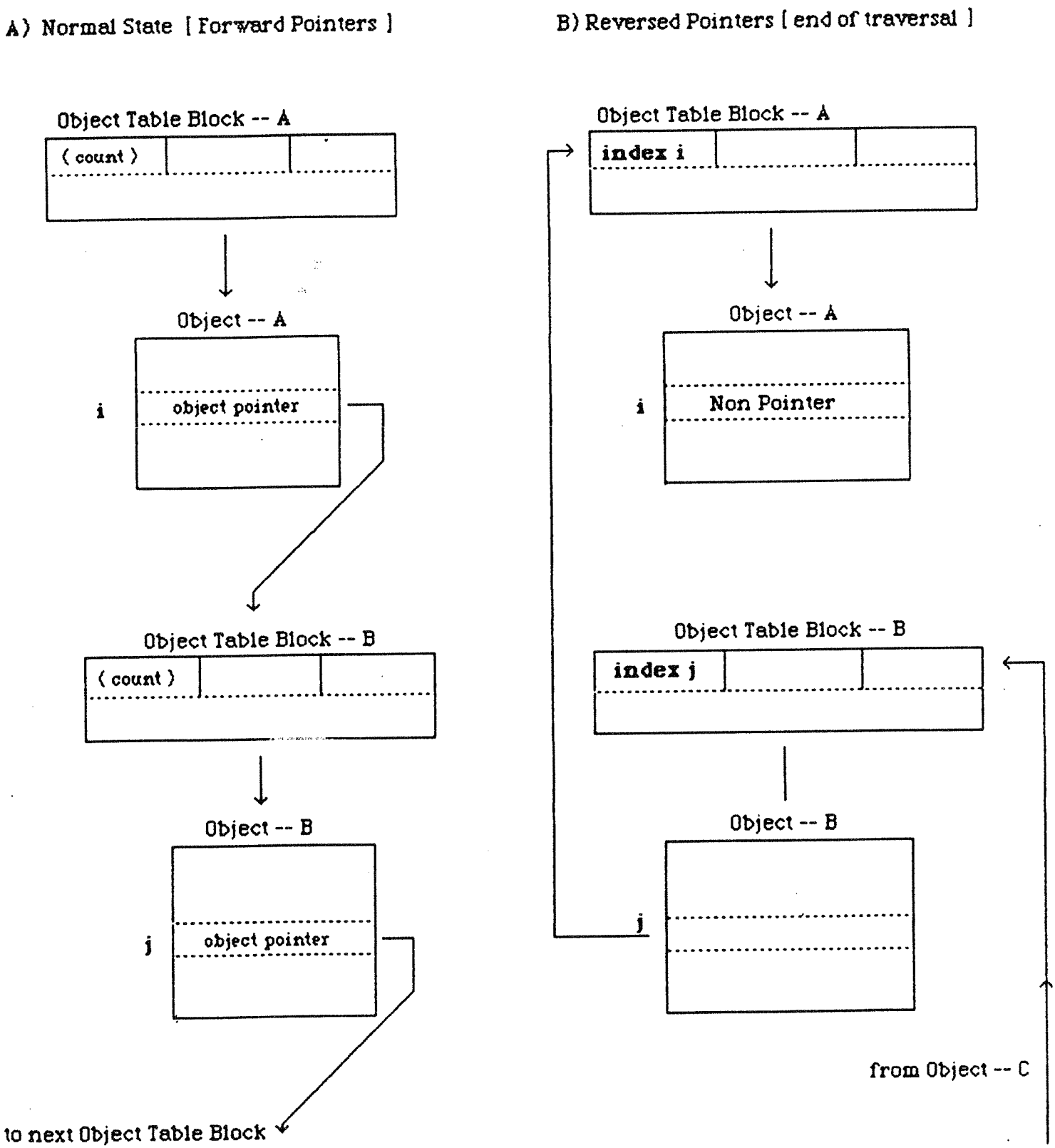
Free Pointer List and Free Chunk List



Non Pointer = 65535

Figure 4:

Pointer Reversal in Nonrecursize Traversal Routine



APPENDIX A: Misprints and Ambiguities

1) Refer to "Part 4: Errata in Smalltalk-80" in the documentation accompanying the tape, SMALLTALK-80: Virtual Image Version 2, for a partial list of misprints in Goldberg and Robson.

2) p. 602 - In description of longUnconditionalJump there is a contradiction. The code description is correct and thus the 11 bit jump displacement is not a 2's complement representation of the the offset for the jump.

3) p. 625 - In lines 4 and 5 of primitiveMakePoint, it should call isIntegerObject instead of isIntegerValue.

4) p. 644 - Goldberg and Robson are ambiguous about where explicit reference count updates are needed to prevent accidental deallocation. These were added to the routines transferTo(p.643), removeFirstLinkOfList(p.644), sleep(p.646), and resume(p.646).

5) p. 647 - Goldberg and Robson does not specify what methodCache should be initialized with. Non Pointers are used in our implementation.

6) p. 688 - Misprint in the specification of isIntegerValue. The routine should read:
^valueWord >= -16384 and: [valueWord < 16384]

7) There is a discrepancy between our tracing and the sample from Xerox. Beginning at cycle = 1680, every time a "16383 @ 16383" point is being created they either get a primitive fail or use a different Bytecode to send the message. However since both the argument and the receiver is within range of the smallinteger, if the range check on the smallinteger is done correctly, there should not be a primitive fail. The effect of this discrepancy is that our version accomplishes the same thing that their's does but takes 16 Bytecodes less. This occurs every time a "16383 @ 16383" message is sent or Bytecode 187 is carried out with the value 16383.

APPENDIX B: Implementation Details

1) Integer division in C truncates (rounds toward zero), so we were forced to create a routine called funkydiv, that always rounds towards negative infinity.

2) In floating-point numbers, the exponent range for Smalltalk is -127 to 128 while range for VAX is -128 to +127. So we use double-precision floating-point numbers in C to preserve range of Smalltalk floating-point numbers. Also, this difference must be accounted for in conversion routine for floating-point numbers.

3) C forces the automatic conversion of various types causing many kludges to be implemented for conversion routines. Consult documentation for conversion routines on p. 688 to get details.

APPENDIX C: Changes from Formal Specification

1) Procedure literal(p.586) is omitted. Instead, a call to literal:ofMethod: is used.

2) Main interpreter loop is modified to allow debugger and tracing.

3) Explicit reference counting added to procedures transferTo(p.643), removeFirstLinkOfList(p.644), sleep(p.646), and resume(p.646) to prevent accidental deallocation.

4) I/O primitives not yet implemented. Currently NoOps.

5) prCoreLeft(p.652) is also currently a NoOp.

6) Heap space is 2^20 contiguous words with the last 64k being object table. All access routines changed to accommodate this lack of segmentation.

7) p. 665 - FreeChunkListHead is a table in the interpreter and not an object in the memory.

8) pp. 676-684 - Both traversal routines for garbage collection and deallocation are open coded to decrease procedure calls, and modified, so as to be called only when actual deallocation occurs.

APPENDIX D: Procedure Reference Table

Smalltalk book Name	C-Name	page
Chapter 27		
fetchInteger: ofObject:	fetchint	574
storeInteger: of object: withValue:	storeint	
transfer:fromIndex:ofObject:toIndex:ofObject:	transfer	
extractBits:to:of:	extractb	575
highByteOf:	highbyte	
lowByteOf	lowbyte	
initializeSmallIntegers	< dcl >	
initializeGuaranteedPointers	< dcl >	576
initializeMethodIndices	< dcl >	577
headerOf	header	
literal:ofMethod:	literal	
temporaryCountOf:	tempcount	
largeContextFlagOf:	largecontext	578
literalCountOf:	litcount	
literalCOuntOfHeader:	litcthd	
objectPointerCountOf:	ptrcount	
initialInstructionPointerOfMethod:	init_ptr	
flagValueOf:	flagval	
fieldIndexOf:	fieldindex	579
headerExtensionOf:	headextension	580
argumentCountOf:	argcount	
primitiveIndexOf:	primindex	
methodClassOf:	methodclass	
initializeContextIndicies	< dcl >	581
instuctionPointerOfContext:	instructptr	582
storeInstructionPointerValue:	store_inst_ptr	
stackPointerOfContext:	stackptr	
storeStackPointerValue:inContext:	storestackptr	583
argumentCountOfBlock:	blockargcount	
fetchContextRegisters	ftch_cntxt_reg	
isBlockContext:	isblockcontext	584
storeContextRegisters	store_reg	
push:	push	585
popstack	popstack	
stacktop	stacktop	

APPENDIX D: Procedure Reference Table

Smalltalk book Name	C-Name	page
stackValue:	stackvalue	
pop:	pop	
unpop:	unpop	585
newActiveContext:	new_act_cntxt	
sender	sender	
caller	caller	586
temporary:	temporary	
literal:	cur_literal	
initializeClassIndicies	< dcl >	587
hash:	hash	
lookupMethodInDictionary:	lookupmethod	588
lookupMethodInClass:	findmthd	589
superClassOf:	superclass	
createActualMessage	createmsg	
initailizeMessageIndices	< dcl >	590
instanceSpecificationOf:	inst_spec	
isPointers:	ispointers	591
isWords:	iswords	
isIndexable:	isindexable	
fixedFieldsOf:	fixedfields	

APPENDIX D: Procedure Reference Table

Smalltalk book Name	C-Name	page
Chapter 28		
fetchbyte		594
interpret		
cycle		
dispatchOn ThisBytecode	dispatch	595
stackbytecode	stackcode	597
pushreceiverVariableBytecode	< combined >	598
pushReceiverVariable:	pushrec	
pushTemporaryVariableBytecode	< combined >	
pushTemporaryVariable:	pushtemp	
pushLiteralConstantBytecode	< combined >	
pushLiteralConstant:	pushlitC	
pushLiteralVariableBytecode	< combined >	
pushLiteralVariable:	pushlitV	599
initializeAssociationIndex	< dcl >	
extendedPushBytecode	extpush	
pushReceiverBytecode	< no routine >	
duplicateTopBytecode	< no routine >	
pushConstantBytecode	pushCon	600
pushActiveContextBytecode	< no routine >	
storeandPopReceiverVariableBytecode	sapRecV	
extendedStoreAndPopBytecode	extsap	
extendedStoreBytecode	extstore	
popstackBytecode	< no routine >	601
jumpBytecode	jumpcode	
jump:	jump	602
shortUnconditionalJump	shrtucjump	
longUnconditionalJump	longucjump	
jumpIf:by:	jumpIf	
sendMustBeBoolean		603
shortConditionalJump	shrtcjump	
longConditionalJump	longcjump	
sendBytecode	sendcode	604
sendLiteralSelectorBytecode	sendlit	
sendSelector:argumentCount:	sendselect	
sendSelectorToClass:	sendto	605
findNewMethodInClass:	findNM	

APPENDIX D: Procedure Reference Table

Smalltalk book Name	C-Name	page
initailizeMethodCache	initmcache	
executeNewMethod	executeNM	
activateNewMethod	activateNM	606
extendedSendBytecode	extsend	
singleExtendedSendBytecode	singleext	
doubleExtendedSendBytecode	doubleext	607
singleExtendedSuperBytecode	singlesuper	
doubleExtendedSuperBytecode	doublesuper	
sendSpecialSelectorBytecode	sendspecial	608
returnBytecode	returncode	
simpleReturnValue:to:	simplereturn	609
returnValue:to:	returnvalue	
returnToAcitveContext	returnTAC	610
nilContextFields	nilCF	

APPENDIX D: Procedure Reference Table

Smalltalk book Name	C-Name	page
Chapter 29		
success: successValue	success	616
success	< a global >	
initPrimitive	initprimitive	
primitiveFail	primitivefail	
popInteger	popinteger	617
pushInteger:	pushinteger	
positive16BitIntegerFor:	p16integer	
positive16BitValueFor:	p16value	
specialSelectorPrimitiveResponse	sspres	618
arithmeticSelectorPrimitive	arithpr	619
commonSelectorPrimitive	commonpr	
primitiveResponse	presp	620
quickReturnSelf	< no routine >	
quickInstanceLoad	quickinstanceload	
dispatchPrimitives	dispatchpr	621
dispatchArithmeticPrimitives	prdisarith	
dispatchIntegerPrimitives	disintegerpr	
primitiveAdd	pradd	622
primitiveDivide	prdivide	
primitiveMod	prmod	623
primitiveDiv	prdiv	
primitiveQuo	prquo	
primitiveEqual	prequal	624
primitveBitAnd	prbitand	
primitiveBitShift	prbitshift	
primitiveMakePoint	prmakepnt	625
initializePointIndices	< dcl >	
dispatchLargeIntegerPrimitives	dislrgintpr	
dispatchFloatPrimitives	disfloatpr	626
dispatchSubscriptAndStreamPrimitives	prdissub	627
checkIndexableBoundsOf:	checkIB	
lengthOf:	lengthof	
subscript:with:	subscript	628
subscript:with:storing:	substore	
primitiveAt	prat	
primitiveAtPut	pratput	629

APPENDIX D: Procedure Reference Table

Smalltalk book Name	C-Name	page
primitveSize	prsize	
primitiveStringAt	prstrat	630
initializeCharacterIndex	< dcl >	
primitiveStringAtPut	prstratput	
initailizeStreamIndices	< dcl >	631
primitveNext	prsubnext	
primitiveNextPut	prsubput	
primitiveAtEnd	pratend	632
dispatchStorageManagementPrimitives	prdisstore	633
primitveObejectAt	probjat	
primitiveObjectAtPut	probjput	634
primitveNew	prnew	
primitiveNewWithArg	prnewargs	
primitiveBecome	prbecome	635
checkInstanceVariableBoundsOf:in:	checkIVB	
primitiveInstVarAt	prinstat	
primitiveInstVarAtPut	prinstput	
primitiveAsOop	prasoop	636
primitiveAsObject	prasobject	
primitiveSomeInstance	prsome	637
primitveNextInstance	prnext	
primitveNewMethod	prnewmethod	
dispatchControlPrimitives	prdiscontrol	
primitiveBlockCopy	prblockcopy	638
primitiveValue	prvalue	639
primitiveValueWithArgs	prvalargs	
primitivePerform	prperform	640
primitivePerformWithArgs	prperargs	641
initializeSchedulerIndices	< dcl >	
asynchronousSignal:	asynch	642
synchronousSignal:	synchsignal	643
transferTo:	to	
checkProcessSwitch	checkPS	
activeProcess	activeprocess	644
schedulerPointer	schedptr	
firstContext	firstcontext	
removeFirstLinkOfList	rem_f_link	
addLastLink:tolist:	addlink	645

APPENDIX D: Procedure Reference Table

Smalltalk book Name	C-Name	page
isemptyList:	isempty	
wakeHighestPriority	wakeHP	
sleep:	sleep	646
suspendActive	suspend	
resume:	resume	
primitiveSignal	prsignal	646
primitiveWait	prwait	
primitiveResume	prresume	647
primitiveSuspend	prsuspend	
primitiveFushCache	prflushcache	
dispatchInputOutputPrimitives	prdisio	
dispatchSystemPrimitives	prdissystem	652
primitiveEquivalent	prequivalent	653
primitiveClass	prclass	
primitiveCoreLeft	prcoreleft	
primitiveQuit	prquit	
primitiveExitToDebugger	prdebug	
primitiveOopsLeft	proopsleft	
primitiveSignalAtOopsLeftWordsLeft	prsigatleft	

APPENDIX D: Procedure Reference Table

Smalltalk book Name	C-Name	page
Chapter 30		
segment:word:	segword	656
segment:word:put:	segwdput	
segment:word:byte:	segbyte	
segment:word:byte:put:	segbyteput	
segment:word:bits:to:	segwbit	657
segment:word:bits:to:put:	segwbitput	
isintegerObject:	< prelim >	660
cantBeIntegerObject:	cantbintobj	661
ot:	ot	
ot:put:	otput	662
ot:bits:to:	otbitsto	
ot:bits:to:put:	otbitput	
countBitsOf:	countbits	
countBitsOf:put:	cntbitsput	
oddBitOf:	oddbit	
oddBitOf:put	oddbitput	
pointerBitOf:	ptrbit	
pointerBitOf:put:	ptrbitput	
freeBitOf:	frbit	
freeBitOf:put:	frbitput	
segmentBitsOf:	segbit	
segmentsBitsOf:put:	segbitput	
locationBitsOf:	locbit	
locationBitsOf:put:	locbitput	663
heapChunkOf:word:	hpchnk	
heapChunkOf:word:put:	hpchnkput	
heapChunkOf:byte:	hpbyte	
heapChunkOf:byte:put:	hpbyteput	
sizeBitsOf:	szbit	
sizeBitsOf:put:	szbitput	
classBitsOf:	clsbit	
classBitsOf:put:	clsbitput	
lastPointerOf:	lastptr	
spaceOccupiedBy:	occspace	
headOfFreePointerList	hdfrptr	665
headOfFreePointerListPut:	hdfrput	

APPENDIX D: Procedure Reference Table

Smalltalk book Name	C-Name	page
toFreePointerListAdd:	addfrptr	666
removeFromFreePointerList	rmfrptr	
headOfFreeChunkList:insegment:	hdchnk	666
headOfFreeChunkList:insegment:put:	hdchnkput	
toFreeChunkList:Add:	tofrchnk	
removeFromFreeChunkList:	rmfrchnk	
resetFreeChunkList:insegment:	resetchnk	667
allocate:class:	< prelim >	668
allocateChunk:	allchunk	
attemptToAllocateChunk:	attchnk	669
attemtToAllocateChunkInCurrentSegment:	attcurchnk	
obtainPointer:location:	obtainptr	670
deallocate:	< prelim >	
abandonFreeChunkInSegment:	abandon	671
releasePointer:	releaseptr	
reverseHeapPointersAbove:	reverse	673
sweepCurrentSegmentFrom:	sweep	
compactCurrentSegment:	compactcurseg	674
countup:	countup	677
countDown:	countdown	
forAllObjectsAccessibleFrom:suchthat:do:	traverse	678
forAllOtherObjectsAccessibleFrom:suchthat:do:	< no routine >	
allocate: extra: class:	< prelim >	679
spaceOccupiedBy:	occspace	680
deallocate:	deallocate	
forAllOtherObjectsAccessibleFrom:suchthat:do:	< no routine >	
reclaimInaccessibleObjects	reclaim	682
zeroReferenceCounts	zero_ref	
markAccessibleObjects	mark_obj	
markObjectsAccessibleFrom:	mark_from	683
rectifyCountsAndDeallocateGarbage	cleanup	
allocateChunk:	allchunk	684
allocate:odd:pointer:extra:class:	allocate	685
lastPointerOf:	lastptr	
spaceOccupiedBy:	occspace	
lastPointerOf:	lastptr	686
fetchPointer:ofObject:	getword	
storePointer:ofObject:withValue:	storeptr	

APPENDIX D: Procedure Reference Table

Smalltalk book Name	C-Name	page
fetchWord:ofObject:	getword	
storeWord:ofObject:withValue:	storeword	
fetchByte:ofObject:	getbyte	687
storeByte:ofObject:withValue:	storebyte	
increaseReferencesTo:	incref	
decreaseReferencesTo:	decref	687
fetchClassOf:	getclass	
fetchWordLengthof:	wordlength	
fetchByteLengthOf:	bytelength	
instantiateClass:withPointers:	instantP	
instantiateClass:withWords:	instantW	
instantiateClass:withBytes:	instantB	
initialInstanceOf:	initialinstance	688
instanceAfter:	instanceafter	
swapPointersOf:and:	swapptrs	
integerValueOf:	int_of	
integerObjectOf:	obj_of	
isIntegerObject:	is_intobj	
isIntegerValue:	is_intval	

BIBLIOGRAPHY

Deutsch, Peter L., "Efficient Implementation of the Smalltalk-80 System", Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, pp. 297-302, 1984.

Goldberg Adele, and Robson, David, SMALLTALK-80: THE LANGUAGE AND ITS IMPLEMENTATION, Addison-Wesley: Reading, Mass., 1983.

Ingalls, Daniel H., "The Smalltalk-76 Programming System Design and Implementation", Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, pp. 9-16, 1978.

Krasner, Glenn, SMALLTALK-80: BITS OF HISTORY, WORDS OF ADVICE, Addison-Wesley: Reading, Mass., 1983.

Rentsch, Tim, "Object Oriented Programming", ACM SIGPLAN NOTICES, Vol. 17, No. 9, pp. 51-57, September 1982.

Schoch, John F., "An Overview of the Programming Language Smalltalk-72", ACM SIGPLAN NOTICES, Vol. 14, No. 9, pp. 64-73, September 1979.

"The Smalltalk System", BYTE, pp. 36-48, August 1981.

SMALLTALK-80: Virtual Image Version 2, Software Concepts Group, Xerox Palo Alto Research Center, 1983.

Suzuki, Norihisa, and Terada, Minoru, "Creating Efficient Systems for Object-Oriented Languages", Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, pp. 290-296, 1984.

Ungar, David M. and Patterson, David A., "Berkeley Smalltalk: Who Knows Where the Time Goes?", SMALLTALK-80: BITS OF HISTORY, WORDS OF ADVICE, Addison-Wesley: Reading, Mass., pp. 189-206, 1983.

Wirfs-Brock, Allen, "Design decisions for Smalltalk-80 Implementors", SMALLTALK-80: BITS OF HISTORY, WORDS OF ADVICE, Addison-Wesley: Reading, Mass., pp. 41-56, 1983.